VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

BEZPEČNOST A OCHRANA SOUKROMÍ V IOT
SECURITY AND PRIVACY IN IOT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Josef Utěkal**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Lukáš Malina, Ph.D.**
SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Josef Utěkal ***ID:*** 174419
Ročník: 3 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Bezpečnost a ochrana soukromí v IoT

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s kryptografickými schématy poskytujícími zabezpečení přenosu dat a ochranu soukromí uživatelů v rámci systémů a aplikací v prostředí IoT. Proveďte analýzu vlastností kryptografických schémat a protokolů, které jsou vhodné i pro omezená zařízení v rámci IoT. Předložte návrh řešení zajišťující bezpečnost datového přenosu a proveďte základní verifikační implementaci schématu na vybrané platformě minipočítače. Cílem bakalářské práce je plně funkční implementace řešení poskytující bezpečnost přenášených dat, testování implementace a měření výpočetní a paměťové náročnosti implementace.

DOPORUČENÁ LITERATURA:

[1] MENEZES, Alfred, Paul C VAN OORSCHOT a Scott A VANSTONE. Handbook of applied cryptography. Boca Raton: CRC Press, c1997. Discrete mathematics and its applications. ISBN 0-8493-8523-7.

[2] KOTHMAYR, Thomas, et al. DTLS based security and two-way authentication for the Internet of Things. Ad Hoc Networks, 2013, 11.8: 2710-2723.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 8.6.2020

Vedoucí práce: doc. Ing. Lukáš Malina, Ph.D.

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

ABSTRAKT

Bakalářská práce se zabývá tématem bezpečnosti a ochrany soukromí v oblasti Internetu věcí (IoT), pro kterou jsou typická zařízení s velmi omezenými výpočetními prostředky. Pro tato výkonově omezená zařízení je důležitý výběr vhodného operačního systému, který by, kromě nízkých výpočetních nároků, nabízel i žádoucí funkcionality pro IoT, např. vykonávání operací v reálném čase. Jako vhodný adept se jeví OS RIOT, který umožní spouštět aplikace napsané v jazyce C na libovolném podporovaném hardwaru – v této práci konkrétně na Arduino Uno a Arduino Due. Práce dále popisuje několik kryptografických knihoven, které jsou podporovány OS RIOT. Cílem je implementace vybraných kryptografických knihoven a měření jejich výpočetní a paměťové náročnosti na obou Arduinech. Tomu předchází teoretické a následně praktické seznámení s OS RIOT, které je realizováno pomocí vytvoření prvního projektu, a měření výpočetní náročnosti jednotlivých kryptografických funkcí knihovny *WolfSSL* na Raspberry Pi 4 Model B.

KLÍČOVÁ SLOVA

Arduino due, Arduino uno, bezpečnost, IoT, Micro-ECC, OS RIOT, Raspberry Pi 4 Model B, Relic, soukromí, WolfSSL

ABSTRACT

The bachelor's thesis deals with the topic of security and privacy in the field of Internet of Things (IoT), for which resource constrained devices are typical. It is very important to find a suitable operating system that, in addition to low computational requirements, would also offer the desired functionality for these constrained IoT devices, such as real-time capabilities. OS RIOT seems to be the one for IoT. It enables you to run applications written in C language on any supported hardware – in this bachelor's thesis specifically on Arduino Uno and Arduino Due. The thesis also describes several cryptographic libraries that are supported by OS RIOT. The aim is to implement selected cryptographic libraries and measure their computational and memory requirements on both Arduino devices. This is preceded by a theoretical and practical familiarization with OS RIOT, which is realized by creating the first project, and measuring the computational complexity of each cryptographic function of the *WolfSSL* library on the Raspberry Pi 4 Model B.

KEYWORDS

Arduino due, Arduino uno, IoT, Micro-ECC, OS RIOT, privacy, Raspberry Pi 4 Model B, Relic, security, WolfSSL

UTĚKAL, Josef. *Bezpečnost a ochrana soukromí v IoT*. Brno, 2020, 42 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: doc. Ing. Lukáš Malina, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Bezpečnost a ochrana soukromí v IoT“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu bakalářské práce panu doc. Ing. Lukáši Malinovi Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Tato bakalářská práce se věnuje bezpečnosti a ochraně soukromí v oblasti Internetu věcí (zkráceně IoT, z anglického Internet of Things), která se v posledních letech těší vysoké popularitě a předpokládá se, že zájem o tato zařízení bude v budoucnu ještě větší. Z tohoto důvodu je zapotřebí myslet na zabezpečení komunikace a ochranu soukromí v IoT. Většinou se jedná o senzory nebo minipočítače, jejichž výkon je oproti běžným počítačům značně omezený, což musíme při návrhu řešení zohlednit.

V první části je popsáno několik typů protokolů, které se k zajištění bezpečnosti a ochrany soukromí v IoT mohou použít. Rozebrány budou konkrétně protokoly DTLS, QUIC a MQTT.

V druhé části budou popsány vlastnosti operačního systému RIOT, který je vhodným OS pro oblast IoT, jelikož byl vytvořen k použití na výkonově omezených zařízeních. OS RIOT bude použit pro implementaci řešení poskytující bezpečnost přenášených dat. Implementace bude nejprve probíhat na výkonnějším zařízení, kterým je v tomto případě Raspberry Pi 4 Model B, kde bude OS RIOT spuštěn jako proces – tzv. native RIOT, který je možný tímto způsobem spustit v distribucích Linuxu nebo na OS X. Fungující řešení se poté přenese i do výkonově omezeného zařízení.

V práci bude ukázán postup vytvoření prvního jednoduchého programu v OS RIOT. Na tomto příkladu budou vysvětleny základní principy vývoje aplikací v RIOTu – nastavení kompilace pro konkrétní podporovaný hardware, definice vlastního příkazu pro příkazový interpret (shell), samotná kompilace a spuštění programu. Dále bude pomocí virtuálních zařízení simulováno spojení mezi dvěma instancemi na linkové vrstvě. Toto seznámení s principy vývoje na OS RIOT bude potřeba pro další práci, kterou bude základní implementace kryptografického schématu.

Dále následuje sekce výkonnostních testů na Raspberry Pi. Pro výkonnostní testy v této části byla zvolena knihovna *WolfSSL*. Tato sekce je základním kamenem pro stanovený cíl v bakalářské práci, kterým bude plně funkční implementace kryptografických knihoven a měření jejich výpočetní a paměťové náročnosti.

Následuje kapitola o kryptografických knihovnách v jazyce C/C++, které jsou vhodné pro omezená zařízení. Bakalářskou práci pak uzavírá kapitola věnující se testům výpočetní a paměťové náročnosti implementovaných knihoven na Arduino Uno/Due.

1 Bezpečnost a ochrana soukromí v IoT

V této části bude popsána bezpečnost IoT. Budou uvedeny různé protokoly, které můžeme v této oblasti použít.

IoT, díky nárůstu počtu chytrých zařízení a zrychlení komunikačních sítí, v poslední době zaznamenává velký nárůst zájmu ze strany veřejnosti v oblasti LLN (Low-Power and Lossy Networks), jež mají výpočetně omezené prostředky. Jedná se o síť, ve které jsou zařízení propojena skrze privátní, nebo veřejné sítě. Tato zařízení při vzájemné výměně informací využívají standardní komunikační protokoly. Při osobním použití je IoT nejčastěji využíváno např. pro vzdálené ovládání spotřebičů v domácnosti či bezpečnostních kamer. Uplatnění ovšem najdeme i ve veřejném sektoru – např. monitorování operací v nemocnicích. Data můžeme pro zefektivnění systému přenášet v reálném čase.

Význam IoT v budoucnu je zjevný vzhledem k jeho současnému užití v každodenních situacích. Vývoj hardwarových technik, jako např. vylepšení šířky pásma zahrnutím kognitivních rádiových sítí pro řešení problému malého využití frekvenčního spektra, napomůže k dalšímu růstu zájmu o oblast Internetu věcí. Bezpečnostní problémy IoT vyvstávají z podstaty užití IP protokolu jako hlavního standardu konektivity. Architektura počítačové platformy musí být zabezpečena před útoky, které mohou narušit funkci daných zařízení, soukromí anebo důvěrnost dat. Kvůli způsobu propojení různých zařízení přebírá IoT bezpečnostní hrozby počítačových sítích, avšak řešení těchto hrozeb musí být uzpůsobeno omezeným prostředkům těchto zařízení. Některá řešení se zaměřují na jednotlivé vrstvy, jiné přístupy řeší zabezpečení typu end-to-end. K zajištění důvěryhodnosti dat je potřeba použít standardizovaný šifrovací algoritmus. Více o bezpečnosti IoT na [1].

1.1 Protokoly pro bezpečnost IoT

V této kapitole bude uvedeno několik možností ochrany pro IoT. Konkrétně se jedná o protokoly Datagram Transport Layer Security (DTLS), Quick UDP Internet Connections (QUIC) a Message Queue Transport Telemetry (MQTT).

1.1.1 Komunikační protokol DTLS

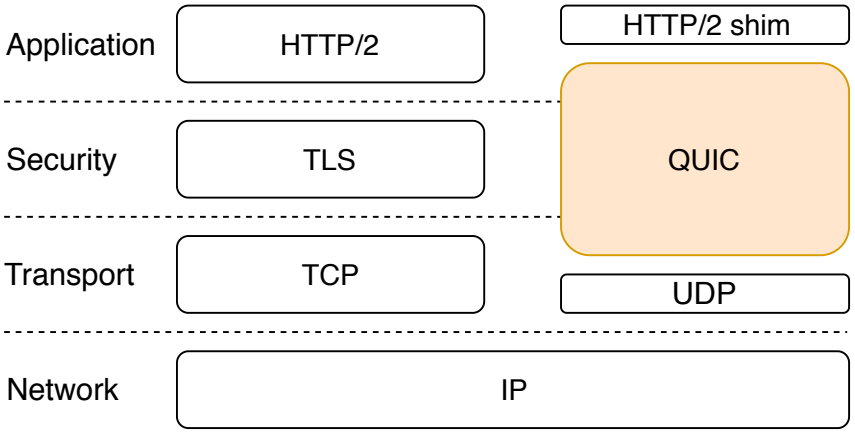
DTLS je příkladem protokolu typu end-to-end, který umožňuje ustanovení oboustranně autentizovaných důvěrných kanálů mezi uzly senzoru a cloudem, a to i v případě nedůvěryhodné síťové infrastruktury. DTLS využívá asymetrickou kryptografii na bázi eliptických křivek k autentizaci dvou koncových zařízení a také k ustanovení

privátních klíčů. Protokol Transport Layer Security (TLS) ve verzi 1.3 byl standardizován organizací Internet Engineering Task Force (IETF) a je považován za vhodný zabezpečovací protokol pro IoT. Problémem však bývá výpočetní složitost protokolu, jelikož IoT zařízení většinou používají jednočipový počítač (mikrokontrolér) s omezenou pamětí. DTLS může být nevhodný i proto, že tato zařízení bývají často napájena bateriemi, od nichž se čeká výdrž v řádu let. Z tohoto hlediska je nutná implementace DTLS, která by měla minimální spotřebu operační paměti a energie.

Protokol DTLS obsahuje dvě hlavní fáze – navazování spojení (handshake) a aplikační data. Ve fázi navazování spojení se klient a server domluví, které kryptografické algoritmy budou používat. Dalším krokem je ustanovení klíčů, jež budou použity k zašifrování následujících zpráv k navázání spojení (handshake messages), protokolem Diffie–Hellman. Dále se klient se serverem navzájem autentizují pomocí digitálních certifikátů a nakonec obě strany potvrdí integritu dat – kvůli možnému Man-in-the-Middle útoku. Tímto je vytvořen důvěryhodný kanál mezi oběma stranami. Tento kanál je nadále využíván ve fázi aplikačních dat k bezpečné výměně šifrovaných dat. Pro více informací o DTLS viz [2].

1.1.2 Zabezpečovací protokol QUIC

QUIC byl navržen k vylepšení výkonu provozu v protokolu Hypertext Transfer Protocol Secure (HTTPS). Dalším parametrem bylo umožnění rychlého zprovoznění v určitém prostředí (např. ve webovém prohlížeči). QUIC se v několika vrstvách liší od současného HTTPS – mimo jiné v použití User Datagram Protocol (UDP) namísto Transmission Control Protocol (TCP) v transportní vrstvě, viz obr. 1.1.

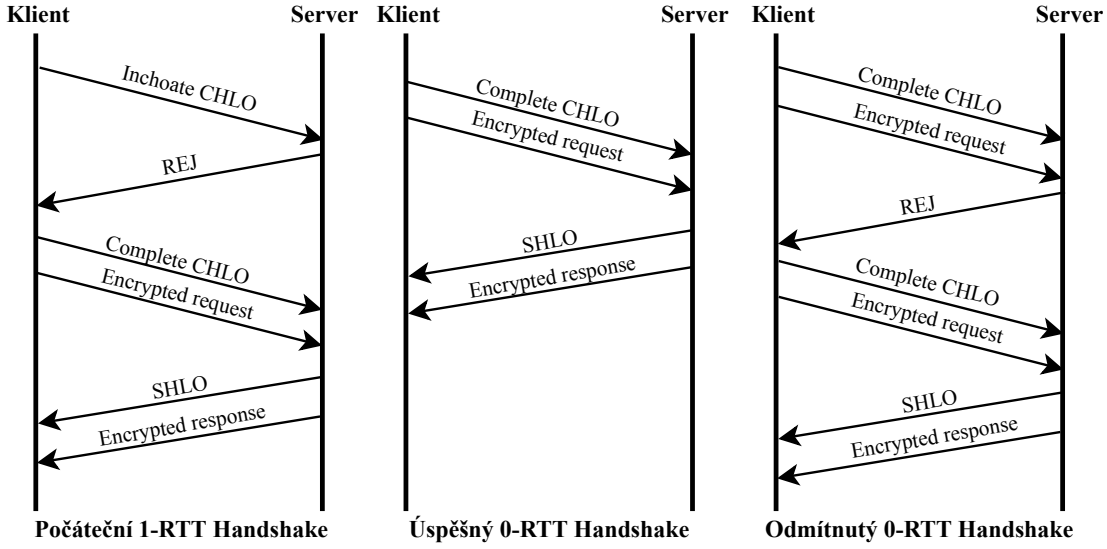


Obr. 1.1: Srovnání tradičního HTTPS a QUIC

Pakety jsou v protokolu QUIC autentizovány a zašifrovány, čímž se zamezuje jejich modifikaci. QUIC používá kryptografický handshake s minimálním zpožděním,

kterého je dosaženo používáním autorizačních údajů známého serveru pro opětovné připojení a rovněž odstraněním nadbytečného místa, které je použito pro režii handshaku (overhead). Užitím datových proudů (streamů) eliminuje zpoždění blokováním čela fronty (Head-of-line blocking), jelikož při ztrátě paketu blokují pouze streamy, které tato data obsahují. Zrychlení, které poskytuje QUIC, můžeme nejvíce pozorovat na serverech s multimediálním obsahem (např. Youtube), kde se čas potřebný k opětovnému uložení obsahu do vyrovnávací paměti snížil o 18 % pro uživatele používající prohlížeč Chrome na počítači a o 15,3 % pro uživatele používající aplikaci na operačním systému Android.

QUIC pro zřízení bezpečného spojení používá kombinaci kryptografického a transportního handshaku. Při úspěšném handshaku si klient uloží informace o serveru – jednotný identifikátor zdroje URI (z anglického Uniform Resource Identifier), jeho název (hostname) a číslo portu – díky čemuž při opětovném připojení ke stejnému serveru nečeká na jeho odpověď a po svém handshaku začne odesílat šifrovaná data. Tomuto způsobu ustanovení spojení se říká Zero Round-Trip Time (0-RTT) – spojení bez obousměrného zpoždění. Na obrázku 1.2 je znázorněno schéma kryptografického handshaku protokolu QUIC včetně zmíněného 0-RTT.



Obr. 1.2: Schéma handshaku v protokolu QUIC

Při prvním připojení nemá klient o serveru žádnou informaci, a tak odešle *inchoate client hello (CHLO)* zprávu, po které následuje *reject (REJ)* zpráva ze strany serveru, která obsahuje nastavení serveru (zahrnut veřejný klíč Diffie–Hellman), certifikát, podpis serveru a token obsahující Internet Protocol (IP) adresu klienta a časové razítko (timestamp). Klient poté při další komunikaci potvrdí svou IP adresu

pomocí tokenu a po přijetí zprávy s nastavením serveru ověří certifikát a podpis serveru. Nakonec odešle *complete CHLO* zprávu obsahující přechodný (ephemeral) klíč klienta. V tomto případě bylo lepší ponechat originální názvosloví, protože po překladu by nemuselo být jisté, o jaké zprávy se jedná. Překlad zprávy reject (odmítnutí) nebo complete (úplná) není problematický, ovšem překlad slova inchoate (jako např. vznikající či v zárodku) může být trochu nevypovídající.

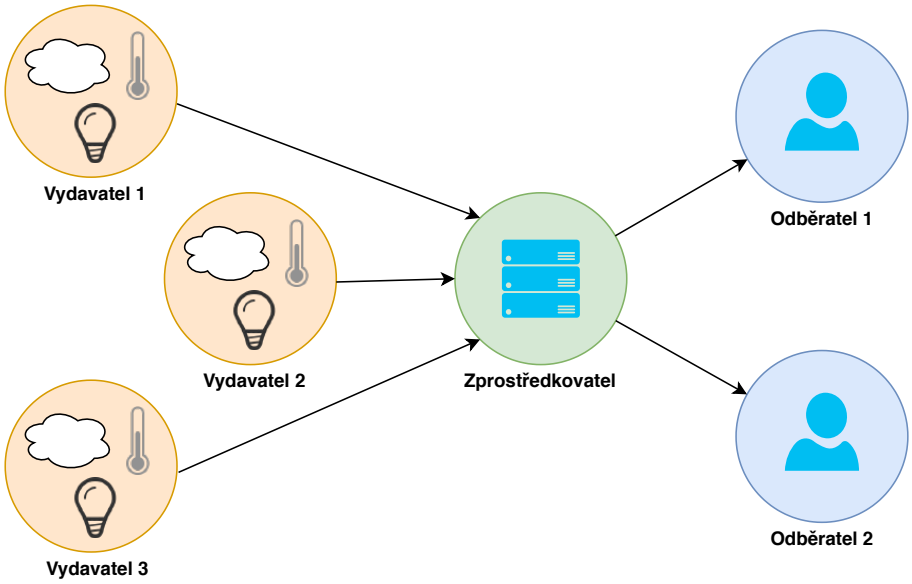
Po odeslání *complete CHLO* může klient posílat data na server. Aby docílil 0-RTT zpoždění, pošle data zašifrovaná svým výchozím klíčem (initial key), a teprve poté čeká na odpověď ze strany serveru. Po úspěšném handshaku server odpovídá zprávou *server hello (SHLO)*, která obsahuje přechodný klíč serveru. V tento okamžik mají obě strany k dispozici dočasný klíč druhé strany, a tak jsou schopny vypočítat konečné (final, nebo forward-secure) klíče, které budou použity k zabezpečení nadcházející komunikace. Jsou tedy použity dvě úrovně utajení – počáteční data klienta jsou šifrována výchozím klíčem, nadcházející klientova data a data serveru jsou šifrována konečnými klíči. Při opakovaném připojení k danému serveru si již klient pamatuje potřebné údaje, a tak již na začátku spojení odesílá *complete CHLO* zprávu následovanou daty zašifrovanými výchozím klíčem, znázorněno uprostřed obr. 1.2. Ustanovení spojení nemusí být úspěšné, např. kvůli vypršení platnosti tokenu nebo změny certifikátu na straně serveru – viz pravé schéma obr. 1.2. V tomto případě server posílá *REJ* zprávu a navázání spojení probíhá jako v případě, kdy klient poslal *inchoate CHLO*.

Dohodnutí o používané verzi probíhá již během navazování spojení. Klient navrhuje verzi v prvním paketu. Pokud server tuto verzi nepodporuje, zašle klientovi zprávu obsahující všechny jím podporované verze, což má za následek obousměrné zpoždění 1-RTT (Round-Trip Time) před navázáním spojení. Další informace o protokolu QUIC naleznete v [3].

1.1.3 Zabezpečení protokolu MQTT

MQTT je vhodným protokolem pro IoT pro svou výpočetní nenáročnost a nízkou spotřebu energie, avšak zaostává v úrovni zabezpečení, které nedosahuje potřebné úrovně. Navíc tato zabezpečení ztrácejí efektivnost ve využití baterie i svou výpočetní nenáročnost. Další velkou výzvou v oblasti bezpečnosti IoT je zabezpečení všech zařízení na síti za běhu, jelikož v aktualizaci nám často brání nespolehlivé sítě, na kterých mnoho těchto zařízení běží.

Prostředky bezpečnosti MQTT je možné rozdělit do několika úrovní, kde každá z úrovní zabraňuje odlišnému typu útoku. Vestavěné zabezpečení v protokolu MQTT je kolekcí bezpečnostních standardů na různých vrstvách, např. v podobě TLS či jeho předchůdce Secure Sockets Layer (SSL) na transportní vrstvě. Vlastní řešení



Obr. 1.3: Schéma protokolu MQTT

zabezpečení není snadné, a zřejmě proto bylo v MQTT při svém vydání použito ověřených standardů. Tyto standardy však nejsou pro IoT příliš vhodné pro svou těžkopádnost. Dále budou popsány různé druhy bezpečnosti, které protokol MQTT obsahuje.

Pro autentizaci MQTT poskytuje možnost zadání uživatelského jména a hesla do paketu *CONNECT*. Uživatelské jméno je textový řetězec (string) kódovaný 8-bit Unicode Transformation Format (UTF-8) a heslo může nabývat délky maximálně 65 535 bajtů. Kromě nemožnosti odeslání hesla bez uživatelského jména, nejsou na heslo kladeny žádné další požadavky. V pokročilejším nastavení má každý klient nastaven unikátní identifikátor (ID) o délce maximálně 65 535 bajtů (jeden znak se rovná jednomu bajtu), který může být, společně s uživatelským jménem a heslem, použit k autentizaci klienta.

K zamezení neoprávněného přístupu k datům slouží autorizace, která je kontrolována na straně zprostředkovatele (broker). MQTT má tyto druhy povolení v režii brokera:

- umožněná témata (topics) – specifická témata, všechna témata,
- umožněné operace – publikovat, odebírat, oboje,
- umožněná kvalita služeb QoS (Quality of Service) – 0, 1, 2, všechny.

Témata můžeme snadno kontrolovat umístěním informací před ID klienta (tzv. prefix). Vzhledem k tomu, že broker není ve většině případů zatížen velkou výpočetní ani energetickou náročností, autorizace může být snadno implementována k potřebám daného použití.

MQTT v základní konfiguraci používá transportní protokol TCP, který komu-

nikaci nešifruje. K šifrování komunikace použijí mnozí brokeři protokol TLS, který ovšem není vhodný pro IoT z důvodů velké spotřeby paměti a výpočetní náročnosti.

Pro větší bezpečnost protokolu MQTT se doporučuje používat klientské certifikáty X.509. Nevýhoda pro tuto zvýšenou bezpečnost spočívá ve vyžadování opatření ze strany klienta a v podpoře mechanismu odebrání certifikátu klienta. Další problém nastává při malém počtu vydavatelů (publisher) k velkému počtu odběratelů (subscriber) – kontrola životních cyklů odběratelských certifikátů je pro výkonnostně omezená zařízení velkou výzvou.

Jako vhodný protokol pro bezpečnou autorizaci, který je, kromě počítače, použitelný i pro mobilní zařízení a webové aplikace, se jeví OAuth ve verzi 2.0, který umožňuje aplikacím třetích stran omezený přístup ke službě Hypertext Transfer Protocol (HTTP). Jeho navržení pro HTTP nicméně znamená, že pro MQTT není vhodným kandidátem.

Šifrování datového obsahu (payload) je v MQTT možné jako součást paketů *PUBLISH*. Payload bude vždy zašifrován na straně vydavatele, ale k jeho dešifrování může dojít jak na straně zprotředkovatele, tak i na straně odběratele – poté by se jednalo o šifrování typu End-to-End. U obou možností lze využít symetrické nebo asymetrické kryptografie, ale zašifrován bude pouze payload. Ostatní data (např. certifikáty nebo informace o odběrateli) zůstávají v otevřené podobě. I v tomto případě se ale může jednat o zátěž, které zařízení IoT nejsou schopna. Dále je třeba zajistit bezpečné ustanovení klíčů. Bohužel žádné z těchto řešení není schopné zamezit útoku Man-in-the-Middle ani útoku přehráním (replay attack).

Kontrola integrity dat může být přidána k datovému obsahu *PUBLISH* paketů užitím libovolného protokolu pro digitální podpis nebo pomocí kontrolního součtu. Kontrola integrity dat nepřidá další úroveň zabezpečení k již silnému bezpečnostnímu systému, ale je vhodným doplňkem k šifrování zpráv. I kdyby byl útočník schopen dešifrovat a znovu zašifrovat zprávu, jakékoliv pozměnění zprávy by se projevilo nesouladem v kontrole integrity. Pro podrobnější informace o protokolu MQTT včetně návrhu na jeho zabezpečení, který je vhodný i pro výkonově omezená zařízení, viz [4].

2 OS RIOT a bezpečnostní knihovny

2.1 Operační systém RIOT

Jak už bylo v textu několikrát zmíněno, výpočetní možnosti IoT zařízení jsou značně omezené, a tak nemohou běžet na obvyklých operačních systémech, jako např. Linux nebo Windows. Jedním z význačných operačních systémů navržených přímo pro nevýkonná IoT zařízení je RIOT, který je spustitelný i na zařízeních s opravdu malou operační pamětí (RAM, z anglického Random Access Memory). RIOT je otevřený (open source) operační systém, který nevyžaduje ani jednotku správy paměti (MMU, z anglického Memory Management Unit) ani jednotku ochrany paměti (MPU, z anglického Memory Protection Unit). Srovnání nároků operačních systémů je znázorněno v tabulce 2.1 převzaté z [5].

Tab. 2.1: Srovnání stávajících operačních systémů

OS	min. RAM	min. ROM	C	C++	multi-threading	MCU bez MMU	modularita	real-time
Contiki	< 2 kB	< 30 kB	•	✗	•	✓	•	•
Tiny OS	< 1 kB	< 4 kB	✗	✗	•	✓	✗	✗
Linux	~ 1 MB	~ 4 MB	✓	✓	✓	✗	•	•
RIOT	$\sim 1{,}5$ kB	~ 5 kB	✓	✓	✓	✓	✓	✓

✓ = plně podporováno, • = částečně podporováno, ✗ = nepodporováno

Malá paměťová náročnost není jedinou předností OS RIOT. Další vlastností umožňující nasazení RIOT na co největší počet zařízení je jeho schopnost pracovat se širokou škálou architektur – od 8 do 32 bitů. Neméně důležitou vlastností operačních systémů v oblasti IoT je podpora multiplatformního hardwaru. Některé operační systémy jsou svázány s hardwarem od konkrétního výrobce a zaměřují se pouze na jednu architekturu. Na rozdíl od řešení, která nabízejí pouze určité prvky operačních systémů, RIOT obsahuje vše, co je očekáváno od plnohodnotného systému – např. hardwarovou abstrakci a systémové knihovny.

2.1.1 Požadavky na vhodný operační systém pro IoT

Je třeba rozlišovat rozdíly v druzích IoT zařízení. Zařízení nižší třídy mívají oproti těm výkonným (např. Raspberry Pi) až milionkrát menší operační paměť, tisíckrát menší kapacitu centrální procesorové jednotky (CPU, z anglického Central Processing Unit), spotřebují tisíckrát méně energie a používají sítě, jejichž propustnost je stotisíckrát menší. Tato zařízení nižší třídy jsou založena na třech hlavních komponentech.

1. Mikrokontrolér (MCU, z anglického Microcontroller Unit) – hardware obsahující CPU, RAM o velikosti několika kB, Read-Only Memory (ROM) a také mapované periférie.
2. Externí zařízení různého druhu, např. senzory, úložiště, výkonné prvky (actuator), které jsou připojeny k mikrokontroléru přes různé vstupní/výstupní (I/O, z anglického input/output) standardy.
3. Alespoň jedno síťové rozhraní pro připojení k Internetu, obvykle využívající přenosovou technologii s nízkou spotřebou energie. Tyto transceivery mohou být součástí MCU nebo jsou připojeny jako externí zařízení.

Mikrokontroléry se podstatně liší v závislosti na výrobci. Mohou být rozdílné, i když jsou navrženy pro stejnou architekturu CPU. Některé vlastnosti jsou však společné:

- mají jedno jádro s taktem v řádu několika MHz,
- neposkytují pokročilé funkce, jako např. MMU.

Hlavními požadavky na chování vhodného operačního systému pro IoT zařízení nižší třídy jsou:

- efektivní využití paměti,
- efektivní spotřeba energie,
- reaktivita.

První bod vychází z velmi omezených velikostí pamětí RAM a ROM, kterými tyto přístroje disponují. Dále se očekává několikaletá životnost na baterii, a proto musí OS co nejvíce využít nízkoenergetických režimů dostupných na hardwaru. V jistých situacích potřebujeme, aby zařízení reagovala v téměř reálném čase – např. pokud je zařízení určeno ke spuštění alarmu nebo jej chceme vzdáleně ovládat.

Je třeba vzít na vědomí, že vylepšením výkonu v jedné oblasti může dojít ke zhoršení výkonu v druhé. Zefektivnění užití paměti může mít za následek více operací na CPU, více kopírování, a tím pádem snížení efektivity spotřeby baterie. Na druhou stranu užití režimu spánku pro vylepšení životnosti baterie má za následek zhoršení reakční doby systému.

Očekává se, že IoT zařízení jsou připojena k síti. Na spojové vrstvě nalezneme několik bezdrátových řešení s nízkou zátěží baterie – např. IEEE 802.15.4, Long Range (LoRa) či Bluetooth low-energy (BLE). Pro drátová připojení to jsou třeba Ethernet a BACnet. OS by měl nabídnout připojení k Internetu pomocí 6LoWPAN, IPv6, UDP a CoAP. Pro směrování paketů nebo ovládání dalších zařízení by vyžadovalo podporu dalších protokolů.

Tím, že existují různorodé druhy hardwaru, které se navíc rychle vyvíjí, by mělo být co nejméně nepřenosného softwaru pro IoT. V tomto ohledu je důležitá podpora hardwarové abstrakce ze strany operačního systému, která zajistí přenositel-

nost většiny kódu na všechen hardware podporovaný OS. Na aplikační úrovni je od OS očekáváno poskytnutí rozhraní pro připojení softwarových modulů třetích stran. Od IoT se očekává, že prorazí jak v soukromé oblasti, tak i v oblasti průmyslové, což je jeden z důvodů zvyšující se poptávky po bezpečnosti a ochraně soukromí těchto systémů. Důležitý je výběr vhodných kryptografických primitiv a jejich následné přizpůsobení přístrojům s omezeným výkonem, které nemají dostatek výkonu pro použití asymetrické kryptografie v plném rozsahu. Pro ochranu soukromí je klíčová transparentnost – vhodným přístupem je použití otevřeného (open source) softwaru, u kterého si můžeme způsob nakládání s citlivými informacemi ověřit ze zdrojového kódu.

2.1.2 Principy OS RIOT

RIOT je otevřený OS založený na modulární architektuře postavené na minimalistickém kernelu. Na jeho vývoji se podílí komunita vývojářů, jejíž hlavní motivací bylo stvoření systému, který:

1. minimalizuje užití pamětí RAM a ROM i energie,
2. podporuje mikrokontroléry architektury 8–32 bitů a několik druhů základních desek,
3. snižuje duplicitu kódu napříč různými konfiguracemi,
4. zajišťuje přenositelnost většiny kódu napříč podporovaným hardwarem,
5. poskytuje platformu k snadnému vývoji dalšího softwaru,
6. umožňuje vykonávání operací v reálném čase.

Výše zmíněné cíle vytvořily principy OS RIOT:

1. Síťové standardy – zaměření na otevřené síťové standardy (např. protokoly IETF).
2. Systémové standardy – podpora významných standardů, jako např. ANSI C, pro širokou podporu aplikací třetích stran. Užití jazyka C zajišťuje lehkou programovatelnost a nízké nároky na prostředky.
3. Jednotné rozhraní pro programování aplikací (API, z angl. Application Programming Interface) – konzistentní API na veškerém podporovaném hardwaru zaručující přenositelnost a minimální duplicitu kódu.
4. Modularita – vytvoření stavebních bloků, které mohou být libovolně kombinovány pro všestranné užití.
5. Statická paměť – použití předrozdělených struktur z důvodů spolehlivosti, zjednodušené validace a verifikace a také kvůli požadavkům na zpracování v reálném čase.
6. Nezávislost na výrobci a technologii – knihovny výrobce nejsou použity, čímž se opět zabraňuje duplicitě kódu a také možnému proprietárnímu uzamčení

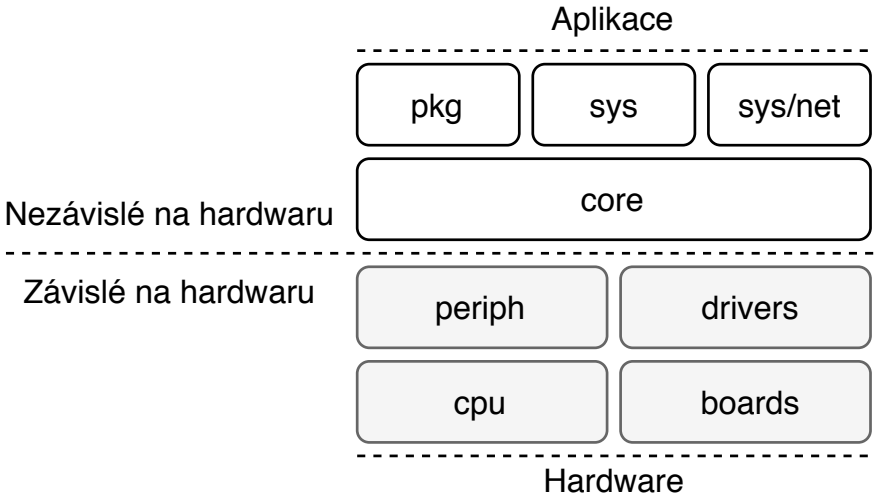
(vendor lock-in).

7. Otevřenost systému – RIOT má v úmyslu nadále zůstat volně dostupným a otevřeným systémem pro každého.

2.1.3 Struktura OS RIOT

RIOT je členěný do softwarových modulů, ze kterých se stane celek při kompilaci a obalí kernel, který poskytuje pouze základní funkcionalitu. Díky tomuto způsobu můžeme použít pouze moduly, které potřebujeme, čímž vytvoříme systém pro specifické užití. Docílíme tak komplexnosti systému a zároveň minimalizujeme spotřebu paměti. Kód OS RIOT je strukturovaný podle skupin, které jsou znázorněny na obrázku 2.1.

1. *core* implementuje kernel a jeho základní datové struktury, jako např. lineární seznamy nebo zásobníky,
2. hardwarová abstrakce rozeznává čtyři části:
   - *cpu* – implementuje funkce související s MCU,
   - *boards* – převážně vybírají, konfigurují a mapují CPU a ovladače,
   - *drivers* – implementují ovladače,
   - *periph* – zajišťuje jednotný přístup k periferiím MCU a je použitý ovladači,
3. *sys* implementuje systémové knihovny, které nespadají do funkcí kernelu – kryptografické knihovny, síťování a podpora souborového systému,
4. *pkg* importuje komponenty třetích stran, tj. knihovny nezahrnuté v hlavním repozitáři kódu,
5. aplikace implementují logiku skutečného druhu použití.



Obr. 2.1: Struktura OS RIOT

2.1.4 Kernel OS RIOT

Kernel OS RIOT se vyvinul z projektu FireKernel. Poskytuje základní funkcionalitu pro plánování procesů, vícevláknovou změnu kontextu (multi-threading context switching), meziprocesovou komunikaci (IPC, z anglického Inter-Process Communication) a synchronizační prostředky (např. algoritmus vzájemného vyloučení). Jak bylo zmíněno výše, ostatní komponenty, jako např. ovladače, jsou uchovány mimo kernel. Vzájemná komunikace komponentů je realizována přes API dodané kernelem.

Vlákno RIOTu se podobá linuxovému vláknu. Každý komponent (např. ovladač pro síťový transceiver) může běžet v jiném kontextu vlákna (thread context) s určitou prioritou. Schopnost vykonat více procesů (multithreading) byla vestavěna pro poskytnutí těchto výhod:

- přehledné logické rozdělení mezi mnohými úkoly,
- snadné stanovení priorit pro úkoly,
- jednodušší import kódu.

Použití multithreadingu není v OS RIOT povinné, což oceníme v případech, kdy je nutná co nejmenší spotřeba paměti – pak je žádoucí použití pouze jednoho vlákna. Aplikace uživatele tak může být jediným vláknem běžícím v systému za předpokladu, že vybraný modul systému nepotřebuje spustit žádné vlákno. Tímto výrazně snížíme paměťové nároky plánovače (scheduler), a tak můžeme vytvořit nesmírně paměťově efektivní firmware na podobném principu jako Arduino, avšak můžeme dále využívat výhod plynoucích z použití OS RIOT (např. ovladače zařízení nebo hardwarová abstrakce).

Kernel RIOTu používá plánovač založený na stálých prioritách a opatřeních s operacemi vykonatelnými v konstantním čase, čímž umožňuje vykonávání úloh v reálném čase s přibližnými zárukami (soft real time). To znamená, že čas potřebný k přerušení jednoho vlákna a přepnutí na druhé nepřekročí určenou horní hranici, přičemž uložení kontextu (context saving), obnovení kontextu (context restoring) a nalezení dalšího vlákna ke spuštění jsou operace deterministické (s pevně stanovenými časy k vykonání). Plánovač pro správnou funkčnost pozdrží vykonávání úloh s nízkou prioritou pro přednostní vykonání událostí s vyšší prioritou.

Jestliže událost vyžaduje akci od vlákna s vyšší prioritou, jsou vlákna s nižší prioritou pozdržena do doby, než se událost dokončí. Aby se minimalizoval čas zpracování a spotřeba energie, nepřepíná se mezi úkoly se stejnou prioritou. Tímto je umožněna reakce systému v reálném čase – priority úkolů jsou uceleně nakonfigurovány. Plánovač použitý v RIOTu není závislý na periodickém tiku CPU, a tak se systém probouzí jen, když se něco děje (např. přerušení vyvolané hardwarem). Probuzení může dále vyvolat například stisknutí kláves nebo přijetí paketu transceiverem. Když neběží žádná vlákna a ani nejsou další pozastavena, systém přepne na

nečinné vlákno (idle thread) mající nejnižší prioritu, které automaticky přepne do nejúspornějšího módu a tím optimalizuje spotřebu energie. Pro detailnější informace o OS RIOT viz [6].

2.2 Kryptografické knihovny podporované OS RIOT

OS RIOT podporuje mnoho externích knihoven a aplikací, mezi nimiž jsou i knihovny implementující kryptografická primitiva. Několik příkladů těchto knihoven bude uvedeno v této sekci.

1. *C25519* – knihovna obsahující implementace D. J. Bernsteinovy křivky *Curve255191* pro výměnu klíče nad eliptickými křivkami v protokolu Diffie–Hellman (ECDH, z angl. Elliptic Curve Diffie–Hellman) a *Ed25519* pro digitální podpis uzpůsobené pro zařízení s malou pamětí. Spotřeba paměti je dostatečně malá pro užití na většině MCU. Obzvlášť násobení skalárem v *Curve255191* využívá méně než 0,5 kB zásobníku.
2. *Micro-ECC* je knihovna využívající kryptografii na bázi eliptických křivek – Elliptic Curve Digital Signature Algorithm (ECDSA) pro digitální podpis a výše zmíněný ECDH pro ustanovení klíče mezi dvěma stranami i po nezabezpečeném kanálu.
3. *Relic* – stále se vyvíjející knihovna, která nabízí široké spektrum kryptografických algoritmů.
4. *TinyDTLS* – knihovna od Eclipse podporující sady šifer (cipher suites) `TLS_PSK_WITH_AES_128_CCM_8` a `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8`. Obě sady používají protokol TLS se 128bitovou šifrou Advanced Encryption Standard (AES), avšak druhá zmíněná užívá protokolů ECDH a ECDSA namísto předsdíleného klíče (PSK, z angl. Pre-shared key).
5. *TweetNaCl* získala název pro svou schopnost vejít se s kódem obsahujícím všechny funkce knihovny na 100 příspěvků na twitteru. Jedním z autorů je D. J. Bernstein, a proto knihovna obsahuje ECDH s *Curve255191* a *Ed25519* pro digitální podpis. Více o TweetNaCl na [7].
6. *WolfSSL* poskytuje vestavěnou knihovnu SSL/TLS vyvinutou pro paměťově omezená zařízení.
7. *qDSA* – zajišťuje paměťově nenáročné a bezpečné digitální podpisy pomocí dvojic klíčů protokolu ECDH.

Více informací o knihovnách dostupných pro OS RIOT naleznete na [8].

3 Výkonnostní testy na RIOT/Raspberry Pi

3.1 První projekt v OS RIOT

V této sekci bude popsán postup k vytvoření prvního projektu v native RIOT a bude ukázána simulace spojení mezi dvěma instancemi. K funkčnosti následujících kroků je potřeba instalace balíčku *bridge-utils*. Dále je potřeba stáhnout klon repozitáře OS RIOT příkazem

```
git clone https://github.com/RIOT-OS/RIOT RIOT
```

V tento moment můžeme vytvořit svůj první projekt. K tomu nám jako šablona poslouží projekt *default* ze složky *RIOT/examples/*, který si zkopírujeme a upravíme ho pro naše užití.

```
cd RIOT/examples
cp -R default prvni_projekt
cd prvni_projekt
```

Ze složky *prvni_projekt* jsme schopni pomocí příkazu `make` zkompilovat jak naši aplikaci, tak i OS RIOT. Otevřeme soubor *Makefile* a upravíme řádek

```
APPLICATION = default
```

který určuje název našeho projektu i spustitelného souboru s naší aplikací, na

```
APPLICATION = prvni_projekt
```

Základní deska, pro kterou chceme náš projekt kompilovat, se nastaví na řádku

```
BOARD ?= native
```

Ponecháme jej beze změny, jelikož nejprve otestujeme aplikaci na Raspberry Pi, které spouští RIOT jako native, tedy jako proces v linuxovém systému. Případnou změnou tohoto řádku uzpůsobíme aplikaci konkrétnímu podporovanému hardwaru, na kterém ji budeme chtít spustit. Aby se mohl RIOT spustit jako proces, potřebuje kompilátor vědět, kde se nachází jeho zdrojový kód, a proto řádek

```
RIOTBASE ?= \$(CURDIR)/../..
```

obsahuje absolutní cestu k hlavnímu adresáři RIOT, který se nachází o dvě úrovně výše, než je adresář našeho projektu. Upravený soubor *Makefile* je zachycen na obrázku 3.1.

```
# name of your application
APPLICATION = prvni_projekt

# If no BOARD is found in the environment, use this default:
BOARD ?= native

# This has to be the absolute path to the RIOT base directory:
RIOTBASE ?= $(CURDIR)/../..
```

Obr. 3.1: Upravený soubor *Makefile*

Hlavní částí souboru *main.c* jsou dva řádky, které po spuštění zkompilovaného projektu spustí shell:

```
char line_buf[SHELL_DEFAULT_BUFSIZE];
shell_run(NULL, line_buf, SHELL_DEFAULT_BUFSIZE);
```

K rozšíření výchozích příkazů vlastními je třeba předat strukturu (struct) s názvem, popisem a ukazatelem vlastním funkcím. Pro demonstraci si vytvoříme funkci, která pozdraví uživatele. Nadále v *shell_run* nahradíme první argument NULL polem námi definovaných příkazů pro shell, ve kterém každý příkaz obsahuje název příkazu, jeho popis, volanou funkci a také řádek obsahující samé NULL pro ukončení seznamu příkazů. Výsledný kód můžeme vidět na obrázku 3.2.

Pro vyzkoušení síťových funkcí si vytvoříme virtuální rozhraní, která budou použita jednotlivými instancemi OS RIOT ke vzájemné komunikaci pomocí tzv. tap zařízení, což jsou virtuální zařízení kernelu sloužící k simulaci spojení na linkové vrstvě. K povolení tohoto spojení je potřeba vytvořit tapbridge, který daná tap zařízení propojí. Tapbridge vytvoříme spuštěním následujícího skriptu, který ustaví spojení mezi dvěma tap zařízeními

```
../../dist/tools/tapsetup/tapsetup -c
```

Pokud bychom chtěli vytvořit více než dvě zařízení, přidáme požadovaný počet na konec tohoto příkazu. Kdybychom naopak chtěli zařízení smazat, napíšeme následující příkaz:

```
../../dist/tools/tapsetup/tapsetup -d
```

Pro kompilaci našeho projektu se musíme nacházet ve složce *prvni_projekt*, ve které v příkazové řádce napíšeme `make`. Kompilátor načte potřebné informace k sestavení spustitelného souboru ze souboru *Makefile*. Po kompilaci bychom měli vidět výpis podobný tomu na obrázku 3.3.

Nyní již můžeme projekt spustit pomocí příkazu

```
./bin/native/prvni_projekt.elf tap0
```

```c
static int greeting(int argc, char **argv) {
    /* Suppress compiler errors */
    (void)argc;
    (void)argv;
    printf("Zdravím Vás a přeji pevné nervy s OS RIOT :)\n");
    return 0;
}

const shell_command_t shell_commands[] = {
    {"greeting", "Pozdravení", greeting},
    { NULL, NULL, NULL }
};

int main(void)
{
#ifdef MODULE_NETIF
    gnrc_netreg_entry_t dump = GNRC_NETREG_ENTRY_INIT_PID(GNRC_NETREG_DEMUX_CTX_ALL,
                                                          gnrc_pktdump_pid);
    gnrc_netreg_register(GNRC_NETTYPE_UNDEF, &dump);
#endif

    (void) puts("Vítejte v OS RIOT!");

    char line_buf[SHELL_DEFAULT_BUFSIZE];
    shell_run(shell_commands, line_buf, SHELL_DEFAULT_BUFSIZE);

    return 0;
}
```

Obr. 3.2: Zdrojový kód v *main.c*

```
   text    data     bss     dec     hex filename
  87081     716   72596  160393   27289 /home/pi/RIOT/examples/prvni_projekt/bin/native/prvni_projekt.elf
```

Obr. 3.3: Vytvoření adresáře *bin* po úspěšné kompilaci

Tímto se spustí instance RIOTu, na které běží *prvni_projekt* jako aplikace ve vláknu linuxového systému. Tato instance je připojena k zařízení *tap0*. V novém okně terminálu můžeme spustit novou instanci, která bude připojena k *tap1*:

```
./bin/native/prvni_projekt.elf tap1
```

Příkaz `help` zobrazí všechny dostupné příkazy, mezi kterými vidíme i námi definovaný příkaz `greeting`, jehož funkčnost je demonstrována na obrázku 3.4.

Pro ověření funkčnosti komunikace mezi oběma instancemi zjistíme fyzickou adresu instance připojené k *tap1* příkazem `ifconfig`, který nám zobrazí i číslo rozhraní. Z druhé instance jí pak pomocí příkazu `txtsnd` odešleme jednoslovnou zprávu. Celý příkaz vypadá takto:

```
txtsnd 4 96:D4:3B:77:07:B4 riot
```

Číslo 4 značí rozhraní cílové instance, následuje její fyzická adresa a příkaz ukončuje samotný vzkaz. Přijetí vzkazu v hexadecimální soustavě můžeme vidět na obrázku 3.5.

```
help
Command              Description
---------------------------------------
greeting             Vypíše pozdrav
reboot               Reboot the node
ps                   Prints information about running threads.
rtc                  control RTC peripheral interface
ifconfig             Configure network interfaces
txtsnd               Sends a custom string as is over the link layer
saul                 interact with sensors and actuators using SAUL
> greeting
greeting
Zdravím Vás a přeji pevné nervy s OS RIOT :)
```

Obr. 3.4: Definovaný příkaz `greeting` mezi výchozími příkazy shellu

```
> PKTDUMP: data received:
~~ SNIP  0 - size:   4 byte, type: NETTYPE_UNDEF (0)
00000000  72  69  6F  74
```

Obr. 3.5: Přijetí zprávy zaslané pomocí `txtsnd`

Informace ohledně vytváření projektů v OS RIOT naleznete na [9] a [10].

3.2 Výkonnostní testy kryptografické knihovny

Pro seznámení s bezpečnostní knihovnou a testy jejich výkonnosti byla pro tuto práci vybrána kryptografická knihovna *wolfSSL*. Prvním krokem je samotné stažení ZIP souboru se zdrojovým kódem z webových stránek této knihovny a následné rozbalení archivu pomocí příkazu `unzip -a`, který zajistí extrahování textových souborů včetně znaků značících ukončení dokumentu (end-of-file characters). K sestavení *wolfSSL* nám na *nix systémech (např. Linux nebo OS X) stačí zadat pouze dva příkazy v kořenové složce *wolfSSL*:

```
./configure
make
```

Pro instalaci *wolfSSL* potřebujeme práva superuživatele. Zadáme příkaz

```
sudo make install
```

K testu, zda jsou kryptografické knihovny *wolfSSL* schopné běžet v sytému, použijeme program *testsuite*. Všechny příklady a testy je zapotřebí spouštět z domovského adresáře *wolfSSL*, aby byly nalezeny veškeré certifikáty a klíče ze složky *certs*. Ke spuštění *testsuite* slouží příkaz

```
./testsuite/testsuite.test
```

Po jeho dokončení by se nám mělo na konci výpisu zobrazit „All tests passed!“, což je známkou, že je vše v pořádku nakonfigurováno a sestaveno. Více k sestavení knihovny *wolfSSL* viz [11].

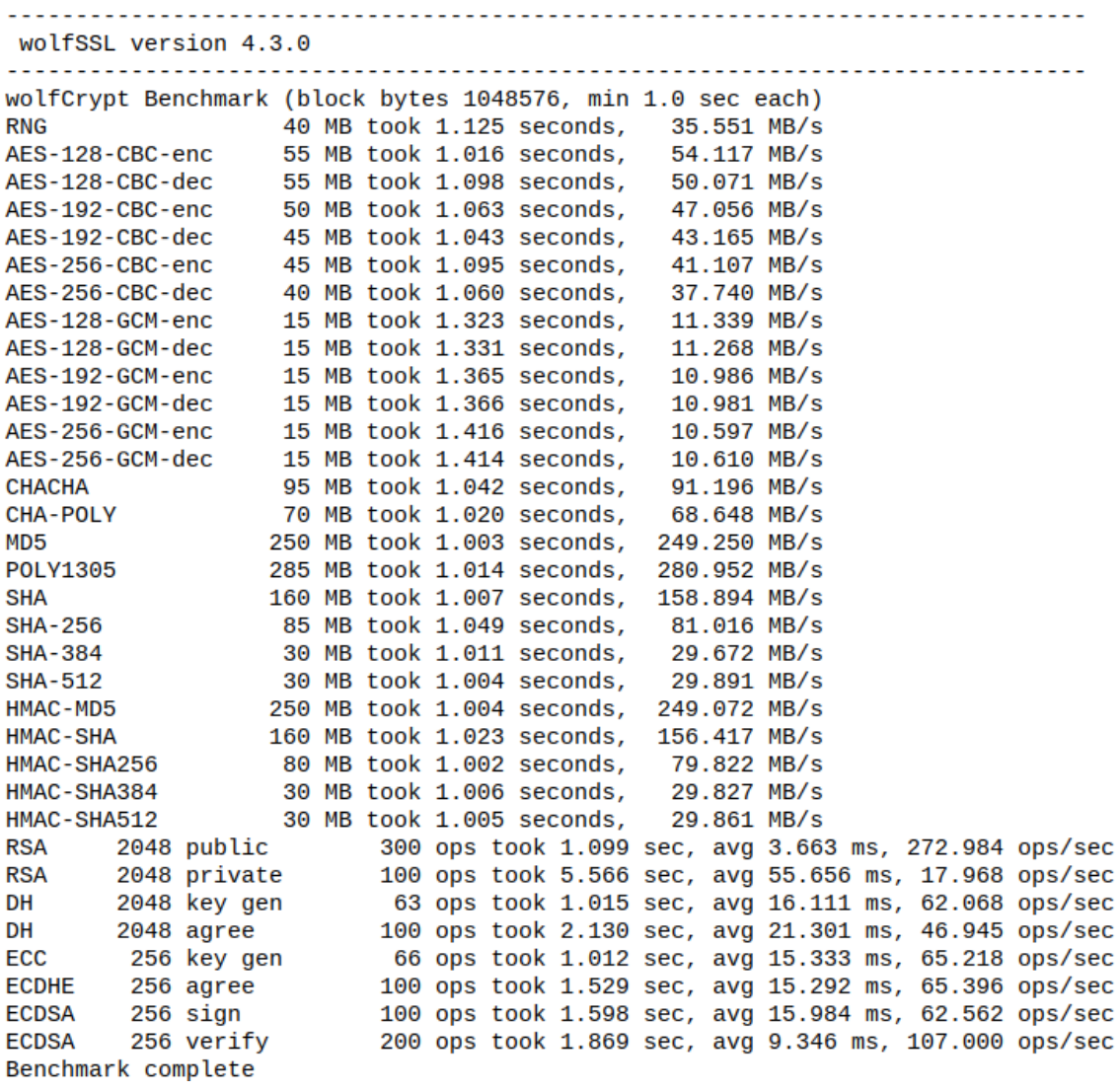
Abychom věděli, jak si dané kryptografické knihovny budou vést na konkrétním hardwaru, můžeme využít velmi užitečného nástroje v podobě srovnávacích testů (benchmark). Bez tohoto nástroje je velmi obtížné hodnotit výkonnost v obecném měřítku, jelikož existuje velké množství různých platforem a kompilátorů.

Aby si mohli uživatelé ověřit výkonnost knihoven na svém zařízení, součástí balíčku *wolfSSL* je aplikace pro benchmark. Protože výkonnost kryptografických knihoven je velmi důležitým aspektem SSL/TLS, tato benchmark aplikace spouští testy výkonnosti použitých algoritmů. Příkaz pro spuštění banchmarku je

```
./wolfcrypt/benchmark/benchmark
```

Výstup tohoto testu můžeme vidět na obrázku 3.6.

Ohledně výkonnostních testů pojednává stránka [12]. Veškeré návody k *wolfSSL* naleznete na [13].



Obr. 3.6: Test kryptografických knihoven balíčku *wolfSSL* na Raspberry Pi

4 Kryptografické knihovny v jazyce C/C++ vhodné pro omezená zařízení

4.1 WolfSSL

WolfSSL je knihovna napsaná v ANSI C, která si zakládá na své výpočetně nenáročné implementaci protokolů TLS a DTLS, a která je zaměřená na výkonově omezená zařízení. Momentálně podporuje TLS 1.3 a DTLS 1.2 při velikosti 20× menší než *OpenSSL*.

Z hašovacích funkcí podporuje např. MD5, SHA2 (SHA-224, SHA-256, SHA-384, SHA-512), SHA3 (BLAKE2), v oblasti asymetrické kryptografie podporuje RSA, DSA, DH. Pro kryptografii nad eliptickými křivkami je zde podpora ECDH, ECDSA. Pro více informací ohledně *WolfSSL* viz [14] a [15].

4.2 Relic

Mezi cíle knihovny *Relic* patří např. snadná přenositelnost a zahrnutí kódu závislého na architektuře, flexibilní konfigurace a maximální efektivnost. Knihovna je členěna do modulů reprezentujících různé skupiny kryptografických metod

Z blokových šifer je implementován AES v režimu řetězení šifrových bloků (CBC, z anglického Cipher Block Chaining). Knihovna dále podporuje např. RSA či Rabin. Z kryptografie nad eliptickými křivkami nechybí ECDH, ECDSA nebo Elliptic Curve Integrated Encryption Scheme (ECIES). Kompletní dokumentace ke knihovně *Relic* je ke stažení na [16].

4.3 Micro-ECC

Micro-ECC je malá a rychlá knihovna implementující ECDH a ECDSA pro 8bitové, 32bitové a 64bitové procesory. Je odolná vůči útokům postranními kanály a podporuje 5 standardních křivek: *secp160r1*, *secp192r1*, *secp224r1*, *secp256r1* a *secp256k1*.

4.4 Doporučené délky klíčů u skupin kryptografických metod

V této podsekci jsou v tabulkách 4.1 a 4.2 uvedeny doporučené velikosti klíčů pro různé skupiny kryptografických metod. Toto doporučení vydává National Institute of Standards and Technology (NIST). Všechny uvedené hodnoty jsou v bitech.

Zkratka FFC pochází z anglického Finite Field Cryptography – kryptografie využívající konečné (Galoisovo) těleso. Patří sem například Digital Signature Algorithm (DSA) nebo protokol Diffie–Hellman (DH). U této skupiny značí „L“ velikost veřejného klíče (public key). Velikost soukromého klíče (private key) je označena písmenem „N“.

IFC je zkratkou z anglického Integer Factorization Cryptography – kryptografie založená na prvočíselném rozkladu. Zástupcem této skupiny je algoritmus Rivest–Shamir–Adleman (RSA).

Poslední zkratkou ECC rozumíme kryptografii na bázi eliptických křivek – Elliptic Curve Cryptography. Do této skupiny patří např. ECDSA nebo ECDH, které byly zmíněny v předchozí sekci. Informace ohledně doporučené délky klíčů naleznete na [15].

Tab. 4.1: Doporučené délky klíčů institutem NIST

bezpečnostní úroveň	symetrické šifry	klíče u FFC	klíče u IFC	klíče u ECC
80	2TDEA, aj.	L = 1024, N = 160	1024	160-223
128	AES-128, aj.	L = 3072, N = 256	3072	256-383
192	AES-192, aj.	L = 7680, N = 384	7680	384-511
256	AES-256, aj.	L = 15360, N = 512	15360	512+

Tab. 4.2: Popis úrovně bezpečnosti

bezpečnostní úroveň	popis
80	bezpečné do roku 2010
128	bezpečné do roku 2030
192	dlouhodobá ochrana
256	bezpečné v předvídatelné budoucnosti

5 Testy knihoven na Arduino Uno/Due

V této kapitole budou testovány výpočetní nároky funkcí kryptografických knihoven na zařízeních, která jsou v oblasti IoT velmi populární – Arduino Uno a Arduino Due. Kromě vzájemného srovnání bude zajímavé pozorovat, zda bude vůbec možné některé kryptografické funkce na Arduino Uno, jež disponuje pouze 2 kB statické paměti (SRAM, z anglického Static Random Access Memory), vykonat. Srovnání základních parametrů obou vývojových desek je možné vidět v tabulce 5.1, převzaté z [17].

Tab. 5.1: Technické specifikace desek Arduino Uno/Due

	Arduino Uno	Arduino Due
MCU	ATmega328P	ATSAM3X8E
rychlost CPU	16 MHz	84 MHz
SRAM	2 kB	96 kB
flash paměť	32 kB	512 kB

Pro měření doby výpočtu bude použit vlastní algoritmus (viz výpis 5.1) využívající funkci *xtimer_now_usec* knihovny *xtimer*. Tato funkce vrací aktuální systémový čas v mikrosekundách. Pro funkčnost je třeba přidat řádek

```
USEMODULE += xtimer
```

do *Makefile* naší aplikace. Více o knihovně *xtimer* naleznete na [18].

Výpis 5.1: Algoritmus pro měření doby výpočtu

```
1  #include <stdio.h>
2  #include <xtimer.h>
3
4  int main(void) {
5    uint32_t time=xtimer_now_usec(); // čas v mikrosekundách
6    // MĚŘENÁ FUNKCE
7    time=xtimer_now_usec()-time; // výsledná doba výpočtu
8    printf("\nCelková doba výpočtu: %ld,%ld ms\n", time/1000, time%1000);
         // pro výsledek v milisekundách dělíme 1000
9    return 0;
10 }
```

5.1 Test knihovny Relic

Při spuštění OS RIOT s aplikací využívající funkci *cp_rsa_gen_quick* knihovny *Relic*, pro výpis jejího kódu viz 5.2, nedojde na Arduino Due k řádnému ukončení. Řádek

```
printf("%d\n",cp_rsa_gen_quick(public, private, 0));
```

který schválně zadává jako délku klíče 0 pro demonstraci, že v tomto případě funkce vrací 1, tedy chybu, ale volání této funkce s délkou klíče libovolné kladné hodnoty náhle ukončí program a řádek

```
printf("\nCelková doba výpočtu: %ld.%ld ms\n", time/1000, time%1000);
```

se již nevypíše. Je tedy možné, že funkce nefunguje správně a je pouze definovaná podmínka pro vrácení chyby, když bude funkce volaná s délkou klíče 0. Kompilátor však nehlásí žádnou chybu. Toto chování je zachyceno na obrázku 5.1.

V této sekci je popsán problém u funkce *cp_rsa_gen_quick*, ale stejný problém nastává i u dalších funkcí, např. *cp_rsa_gen_basic* a *cp_rabin_gen*.

Výpis 5.2: Implementace funkce knihovny *Relic* pro generování klíčů

```
1  #include <stdio.h>
2  #include <xtimer.h>
3  #include <relic.h>
4
5  int main(void) {
6    rsa_t public;
7    rsa_t private;
8
9    uint32_t time=xtimer_now_usec();
10
11   printf("%d\n",cp_rsa_gen_quick(public, private, 0));
12   printf("%d\n",cp_rsa_gen_quick(public, private, 64));
13
14   time=xtimer_now_usec()-time;
15   printf("\nCelková doba výpočtu: %ld.%ld ms\n", time/1000, time%1000);
16   return 0;
17 }
```

```
Device       : ATSAM3X8
Chip ID      : 285e0a60
Version      : v1.1 Dec 15 2010 19:25:04
Address      : 524288
Pages        : 2048
Page Size    : 256 bytes
Total Size   : 512KB
Planes       : 2
Lock Regions : 32
Locked       : none
Security     : false
Boot Flash   : false
Erase flash

Done in 0.049 seconds
Write 23280 bytes to flash (91 pages)
[==============================] 100% (91/91 pages)
Done in 4.485 seconds
Verify 23280 bytes of flash
[==============================] 100% (91/91 pages)
Verify successful
Done in 4.317 seconds
Set boot flash true
CPU reset.
/home/josef/RIOT/dist/tools/pyterm/pyterm -p "/dev/ttyACM0" -b "115200"
2020-06-08 17:11:55,857 # Connect to serial port /dev/ttyACM0
Welcome to pyterm!
Type '/exit' to exit.
2020-06-08 17:11:56,866 # main(): This is RIOT! (Version: 2020.07-devel-684
-g70543b)
2020-06-08 17:11:56,867 # 1
```

Obr. 5.1: Spuštění OS RIOT s *Relic*

5.2 Test knihovny Micro-ECC

Při pokusu o generování klíčů funkcí *uECC_make_key* knihovny *Micro-ECC*, viz výpis 5.3 dojde k chybě „implicitní deklarace funkce *uECC_make_key*“. Chování je zachyceno na obrázku 5.2.

Výpis 5.3: Implementace funkce knihovny *Micro-ECC* pro generování klíčů

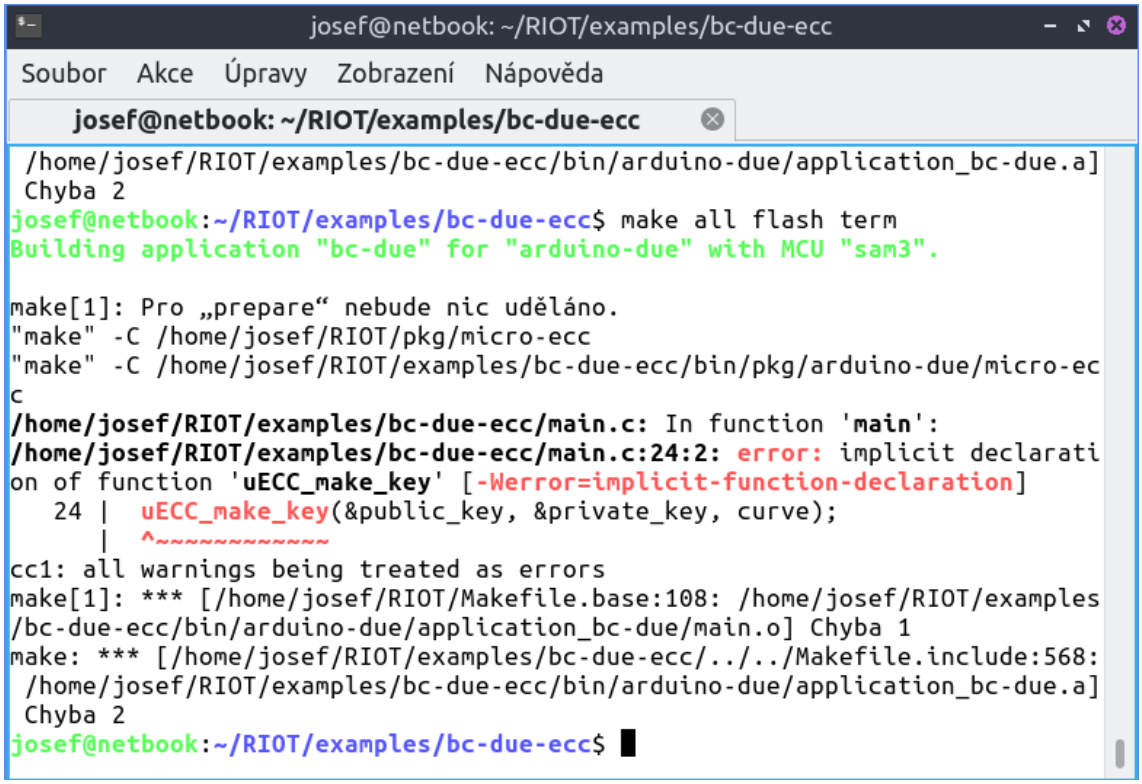
```c
#include <uECC.h>
#include <stdio.h>
#include <xtimer.h>

int main(void) {
  uint8_t public_key[64];
  uint8_t private_key[32];
  const struct uECC_Curve_t * curve = uECC_secp160r1();

  uint32_t time=xtimer_now_usec();

  uECC_make_key(&public_key, &private_key, curve);

  time=xtimer_now_usec()-time;
  printf("\nCelková doba výpočtu: %ld.%ld ms\n", time/1000, time%1000);
  return 0;
}
```



Obr. 5.2: Chyba – implicitní deklarace funkce

Volání jiných funkcí knihovny funguje, viz výpis 5.4. Funkce podle očekávání vrací hodnoty délek klíčů, což je možné vidět na obrázku 5.3.

Výpis 5.4: Funkce knihovny *Micro-ECC* vracející délky klíčů konkrétní křivky

```
1 #include <uECC.h>
2 #include <stdio.h>
3 #include <xtimer.h>
4
5 int main(void) {
6   uint32_t time=xtimer_now_usec();
7
8   printf("%d\n",uECC_curve_public_key_size(uECC_secp256r1()));
9   printf("%d\n",uECC_curve_private_key_size(uECC_secp256r1()));
10
11  time=xtimer_now_usec()-time;
12  printf("\nCelková doba výpočtu: %ld.%ld ms\n", time/1000, time%1000);
13  return 0;
14 }
```

```
Device       : ATSAM3X8
Chip ID      : 285e0a60
Version      : v1.1 Dec 15 2010 19:25:04
Address      : 524288
Pages        : 2048
Page Size    : 256 bytes
Total Size   : 512KB
Planes       : 2
Lock Regions : 32
Locked       : none
Security     : false
Boot Flash   : false
Erase flash

Done in 0.057 seconds
Write 12796 bytes to flash (50 pages)
[==============================] 100% (50/50 pages)
Done in 2.454 seconds
Verify 12796 bytes of flash
[==============================] 100% (50/50 pages)
Verify successful
Done in 2.392 seconds
Set boot flash true
CPU reset.
/home/josef/RIOT/dist/tools/pyterm/pyterm -p "/dev/ttyACM0" -b "115200"
2020-06-08 16:53:38,158 # Connect to serial port /dev/ttyACM0
Welcome to pyterm!
Type '/exit' to exit.
2020-06-08 16:53:39,168 # main(): This is RIOT! (Version: 2020.07-devel-684
-g70543b)
2020-06-08 16:53:39,169 # 64
2020-06-08 16:53:39,169 # 32
2020-06-08 16:53:39,170 #
2020-06-08 16:53:39,173 # Celková doba výpočtu: 0.515 ms
```

Obr. 5.3: Vypsání délek klíčů

Závěr

Tato práce se věnovala oblasti bezpečnosti a ochrany soukromí v IoT. První část seznámila s několika typy protokolů, které se k zajištění bezpečnosti a ochrany soukromí v IoT mohou použít. Konkrétně popsala protokoly DTLS, QUIC a MQTT.

Druhá část práce popsala vlastnosti operačního systému RIOT a vysvětlila, proč je vhodným operačním systémem pro oblast IoT – byl vytvořen k použití na výkonově omezených zařízeních a přitom mu nechybí podpora důležitých funkcí, které známe z běžných operačních systémů. V této práci se s OS RIOT pracovalo na Raspberry Pi 4 Model B, na kterém se operační systém spouští jako proces. Na Raspberry Pi se v bakalářské práci budou testovat aplikace pro výkonově omezené zařízení, na které se následně funkční řešení přenese. Cílem je zabezpečená komunikace mezi Raspberry Pi a dalším, výkonově omezenějším stroji podporovaným RIOTem, který na tomto systému poběží.

Dále byl názorně předveden postup vytvoření první jednoduché aplikace v prostředí OS RIOT. Hlavním cílem bylo seznámení se základními principy tvorby programů v RIOTu, jako např. snadné uzpůsobení aplikace pro cílovou platformu – změna pouze jednoho řádku s informací o základní desce. Byla předvedena kompilace a spuštění programu přes příkazový řádek. Dále se simulovalo spojení mezi dvěma instancemi. Další částí bakalářské práce byly výkonnostních testy knihovny *WolfSSL* na Raspberry Pi.

Následovala kapitola o kryptografických knihovnách v jazyce C/C++, které jsou vhodné pro omezená zařízení. Hlavním cílem bakalářské práce je však plně funkční implementace kryptografických knihoven a měření jejich výpočetní a paměťové náročnosti na Arduino Uno/Due. Tento cíl se však nepodařilo splnit kvůli problémům uvedeným v této závěrečné kapitole.

Literatura

[1] KHAN, Minhaj Ahmad a Khaled SALAH. IoT security: Review, blockchain solutions, and open challenges. *Future Generation Computer Systems* [online]. 2018, **82**, 395–411 [cit. 2019-11-20]. Dostupné z: https://doi.org/10.1016/j.future.2017.11.022

[2] BANERJEE, Utsav, Andrew WRIGHT, Chiraag JUVEKAR, Madeleine WALLER, ARVIND a Anantha P. CHANDRAKASAN. An Energy-Efficient Reconfigurable DTLS Cryptographic Engine for Securing Internet-of-Things Applications. *IEEE Journal of Solid-State Circuits* [online]. 2019, **54**(8), 2339–2352 [cit. 2019-12-01]. DOI: 10.1109/JSSC.2019.2915203. ISSN 1558-173X. Dostupné z: https://ieeexplore.ieee.org/document/8721457/

[3] LANGLEY, Adam, Janardhan IYENGAR, Jeff BAILEY, et al. The QUIC Transport Protocol. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM '17* [online]. New York, New York, USA: ACM Press, 2017, 2017, s. 183–196 [cit. 2019-12-01]. DOI: 10.1145/3098822.3098842. ISBN 978-1-4503-4653-5. Dostupné z: http://dl.acm.org/citation.cfm?doid=3098822.3098842

[4] MALINA, Lukáš, Gautam SRIVASTAVA, Petr DZURENDA, Jan HAJNÝ a Radek FUJDIAK. A Secure Publish/Subscribe Protocol for Internet of Things. In: *Proceedings of the 14th International Conference on Availability, Reliability and Security - ARES '19* [online]. New York, New York, USA: ACM Press, 2019, 2019, s. 1–10 [cit. 2019-12-02]. DOI: 10.1145/3339252.3340503. ISBN 978-1-4503-7164-3. Dostupné z: http://dl.acm.org/citation.cfm?doid=3339252.3340503

[5] *RIOT – The friendly Operating System for the Internet of Things* [online]. 2007 [cit. 2019-12-03]. Dostupné z: https://riot-os.org/

[6] BACCELLI, Emmanuel, Cenk GUNDOGAN, Oliver HAHM, et al. RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. *IEEE Internet of Things Journal* [online]. 2018, **5**(6), 4428–4440 [cit. 2019-12-04]. DOI: 10.1109/JIOT.2018.2815038. ISSN 2327-4662. Dostupné z: https://ieeexplore.ieee.org/document/8315125/

[7] BERNSTEIN, Daniel J., Bernard van GASTEL, Tanja LANGE, Peter SCHWABE a Sjaak SMETSERS. *TweetNaCl: A crypto library in 100 tweets* [online]. 2014 [cit. 2019-12-13]. Dostupné z: https://tweetnacl.cr.yp.to/tweetnacl-20140917.pdf

[8] Packages. *RIOT - The friendly Operating System for the Internet of Things* [online]. [cit. 2019-12-13]. Dostupné z: https://doc.riot-os.org/group__pkg.html#details

[9] Creating your first RIOT project. *The world's leading software development platform · GitHub* [online]. c2019, 17 Jan 2018 [cit. 2019-12-17]. Dostupné z: https://github.com/RIOT-OS/RIOT/wiki/Creating-your-first-RIOT-project

[10] Using and Developing with RIOT. *CodeProject - For those who code* [online]. 2019, 12/19/2016 [cit. 2019-12-18]. Dostupné z: https://www.codeproject.com/Articles/840499/RIOT-Tutorial

[11] WolfSSL User Manual | Chapter 2: Building wolfSSL | Documentation. *WolfSSL Embedded SSL/TLS Library | Now Supporting TLS 1.3* [online]. c2019 [cit. 2019-12-20]. Dostupné z: https://www.wolfssl.com/docs/wolfssl-manual/ch2/

[12] WolfSSL User Manual | Chapter 3: Getting Started | Documentation. *WolfSSL Embedded SSL/TLS Library | Now Supporting TLS 1.3* [online]. c2019 [cit. 2019-12-20]. Dostupné z: https://www.wolfssl.com/docs/wolfssl-manual/ch3/

[13] WolfSSL User Manual | wolfSSL Embedded SSL/TLS Library Docs. *WolfSSL Embedded SSL/TLS Library | Now Supporting TLS 1.3* [online]. c2019 [cit. 2019-12-20]. Dostupné z: https://www.wolfssl.com/docs/wolfssl-manual/

[14] WolfSSL Embedded SSL/TLS Library | wolfSSL Products. *WolfSSL Embedded SSL/TLS Library | Now Supporting TLS 1.3* [online]. c2020 [cit. 2020-05-30]. Dostupné z: https://www.wolfssl.com/products/wolfssl/

[15] WolfSSL User Manual | Chapter 4: Features | Documentation. *WolfSSL Embedded SSL/TLS Library | Now Supporting TLS 1.3* [online]. c2020 [cit. 2020-05-30]. Dostupné z: https://www.wolfssl.com/docs/wolfssl-manual/ch4/

[16] GitHub - relic-toolkit/relic-doc: Documentation. *The world's leading software development platform · GitHub* [online]. c2020 [cit. 2020-05-25]. Dostupné z: https://github.com/relic-toolkit/relic-doc

[17] Arduino - Compare. *Arduino - Home* [online]. c2020 [cit. 2020-05-27]. Dostupné z: https://www.arduino.cc/en/products/compare

[18] Timers. *RIOT - The friendly Operating System for the Internet of Things* [online]. c2013-2020 [cit. 2020-05-27]. Dostupné z: `https://riot-os.org/api/group__sys__xtimer.html`

Seznam symbolů, veličin a zkratek

•	částečně podporováno
✗	nepodporováno
✓	plně podporováno
0-RTT	Zero Round-Trip Time
1-RTT	Round-Trip Time
6LoWPAN	IPv6 over Low-Power Wireless Personal Area Networks
AES	Advanced Encryption Standard
ANSI	American National Standards Institute
API	Application Programming Interface
BLE	Bluetooth Low-Energy
CBC	Cipher Block Chaining
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
DH	Diffie–Hellman
DSA	Digital Signature Algorithm
DTLS	Datagram Transport Layer Security
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie–Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
ECIES	Elliptic Curve Integrated Encryption Scheme
FFC	Finite Field Cryptography
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
CHLO	Client Hello
ID	identifikátor
IETF	Internet Engineering Task Force
IFC	Integer Factorization Cryptography
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-Process Communication
IPv6	Internet Protocol version 6
LLN	Low-Power and Lossy Networks
LoRa	Long Range
MCU	Microcontroller Unit
MD5	Message-Digest algorithm 5
MMU	Memory Management Unit
MPU	Memory Protection Unit

MQTT	Message Queue Transport Telemetry
NIST	National Institute of Standards and Technology
OS	Operating system
PSK	Pre-Shared Key
QoS	Quality of Service
QUIC	Quick UDP Internet Connections
RAM	Random Access Memory
REJ	reject
ROM	Read-Only Memory
RSA	Rivest–Shamir–Adleman
SHA	Secure Hash Algorithm
SHLO	Server Hello
SRAM	Static Random Access Memory
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
UTF-8	8-bit Unicode Transformation Format